# Beej's Guide to Interprocess Communication

Brian "Beej Jorgensen" Hall

# Chapter 2

# A `fork()` Primer

"Fork", aside from being one of those words that begins to appear very strange after you've typed it repeatedly, refers to the way Unix creates new processes. This document gives a quick and dirty `fork()` primer, since use of that system call will pop up in other IPC documents. If you already know all about `fork()`, you might as well skip this document.

## 2.1 "Seek ye the Gorge of Eternal Peril"

`fork()` can be thought of as a ticket to power. Power can sometimes be thought of as a ticket to destruction. Therefore, you should be careful while messing with `fork()` on your system, especially while people are cranking their nearly-late semester projects and are ready to nuke the first organism that brings the system to a halt. It's not that you should never play with `fork()`, you just have to be cautious. It's kind of like sword—swallowing; if you're careful, you won't disembowel yourself.

Since you're still here, I suppose I'd better deliver the goods. Like I said, `fork()` is how Unix starts new processes. Basically, how it works is this: the parent process (the one that already exists) `fork()`'s a child process (the new one). The child process gets a *copy* of the parent's data. *Voila!* You have two processes where there was only one!

Of course, there are all kinds of gotchas you must deal with when `fork()`ing processes or else your sysadmin will get irate with you when you fill of the system process table and they have to punch the reset button on the machine.

First of all, you should know something of process behavior under Unix. When a process dies, it doesn't really go away completely. It's dead, so it's no longer running, but a small remnant is waiting around for the parent process to pick up. This remnant contains the return value from the child process and some other goop. So after a parent process `fork()`s a child process, it must `wait()` (or `waitpid()`) for that child process to exit. It is this act of `wait()`ing that allows all remnants of the child to vanish.

Naturally, there is an exception to the above rule: the parent can ignore the `SIGCHLD` signal (`SIGCLD` on some older systems) and then it won't have to `wait()`. This can be done (on systems that support it) like this:

```
main()
{
    signal(SIGCHLD, SIG_IGN);  /* now I don't have to wait()! */
    .
    .
    fork();fork();fork();  /* Rabbits, rabbits, rabbits! */
```

Now, when a child process dies and has not been `wait()`ed on, it will usually show up in a `ps` listing as "`<defunct>`". It will remain this way until the parent `wait()`s on it, or it is dealt with as mentioned below.

Now there is another rule you must learn: when the parent dies before it `wait()`s for the child (assuming it is not ignoring `SIGCHLD`), the child is reparented to the `init` process (PID 1). This is not a problem if the child is still living well and under control. However, if the child is already defunct, we're in a bit of a bind. See, the original parent can no longer `wait()`, since it's dead. So how does `init` know to `wait()` for these *zombie processes*?

The answer: it's magic! Well, on some systems, `init` periodically destroys all the defunct processes it owns. On other systems, it outright refuses to become the parent of any defunct processes, instead destroying them immediately. If you're using one of the former systems, you could easily write a loop that fills up the process table with defunct processes owned by `init`. Wouldn't that make your sysadmin happy?

Your mission: make sure your parent process either ignores `SIGHCLD`, or `wait()`s for all the children it `fork()`s. Well, you don't *always* have to do that (like if you're starting a daemon or something), but you code with caution if you're a `fork()` novice. Otherwise, feel free to blast off into the stratosphere.

To summerize: children become defunct until the parent `wait()`s, unless the parent is ignoring `SIGCHLD`. Furthermore, children (living or defunct) whose parents die without `wait()`ing for them (again assuming the parent is not ignoring `SIGCHLD`) become children of the `init` process, which deals with them heavy-handedly.

## 2.2 "'I'm mentally prepared! Give me *The Button*!"

Right! Here's an example[1] of how to use `fork()`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
    pid_t pid;
    int rv;

    switch(pid = fork()) {
    case -1:
        perror("fork");  /* something went wrong */
        exit(1);         /* parent exits */

    case 0:
        printf(" CHILD: This is the child process!\n");
        printf(" CHILD: My PID is %d\n", getpid());
        printf(" CHILD: My parent's PID is %d\n", getppid());
        printf(" CHILD: Enter my exit status (make it small): ");
        scanf(" %d", &rv);
        printf(" CHILD: I'm outta here!\n");
        exit(rv);

    default:
        printf("PARENT: This is the parent process!\n");
        printf("PARENT: My PID is %d\n", getpid());
        printf("PARENT: My child's PID is %d\n", pid);
        printf("PARENT: I'm now waiting for my child to exit()...\n");
        wait(&rv);
        printf("PARENT: My child's exit status is: %d\n", WEXITSTATUS(rv));
        printf("PARENT: I'm outta here!\n");
```

[1] https://beej.us/guide/bgipc/source/examples/fork1.c

```
    }

    return 0;
}
```

There is a ton of stuff to note from this example, so we'll just start from the top, shall we?

`pid_t` is the generic process type. Under Unix, this is a `short`. So, I call `fork()` and save the return value in the `pid` variable. `fork()` is easy, since it can only return three things:

| Return Value | Description |
|---|---|
| `0` | If it returns `0`, you are the child process. You can get the parent's PID by calling `getppid()`. Of course, you can get your own PID by calling `getpid()`. |
| `-1` | If it returns `-1`, something went wrong, and no child was created. Use `perror()` to see what happened. You've probably filled the process table—if you turn around you'll see your sysadmin coming at you with a fireaxe. |
| Anthing else | Any other value returned by `fork()` means that you're the parent and the value returned is the PID of your child. This is the only way to get the PID of your child, since there is no `getcpid()` call (obviously due to the one-to-many relationship between parents and children.) |

When the child finally calls `exit()`, the return value passed will arrive at the parent when it `wait()`s. As you can see from the `wait()` call, there's some weirdness coming into play when we print the return value. What's this `WEXITSTATUS()` stuff, anyway? Well, that is a macro that extracts the child's actual return value from the value `wait()` returns. Yes, there is more information buried in that `int`. I'll let you look it up on your own.

"How," you ask, "does `wait()` know which process to wait for? I mean, since the parent can have multiple children, which one does `wait()` actually wait for?" The answer is simple, my friends: it waits for whichever one happens to exit first. If you must, you can specify exactly which child to wait for by calling `waitpid()` with your child's PID as an argument.

Another interesting thing to note from the above example is that both parent and child use the `rv` variable. Does this mean that it is shared between the processes? *NO!* If it was, I wouldn't have written all this IPC stuff. *Each process has its own copy of all variables.* There is a lot of other stuff that is copied, too, but you'll have to read the `man` page to see what.

A final note about the above program: I used a switch statement to handle the `fork()`, and that's not exactly typical. Most often you'll see an if statement there; sometimes it's as short as:

```
if (!fork()) {
        printf("I'm the child!\n");
        exit(0);
    } else {
        printf("I'm the parent!\n");
        wait(NULL);
    }
```

Oh yeah—the above example also demonstrates how to `wait()` if you don't care what the return value of the child is: you just call it with `NULL` as the argument.

## 2.3 Summary

Now you know all about the mighty `fork()` function! It's more useful that a wet bag of worms in most computationally intensive situations, and you can amaze your friends at parties. I swear. Try it.